Automated Behavior Computation for Software Analysis and Validation

Mark Pleszkoch, Richard Linger,
Stacy Prowell, and Kirk Sayre
Cyber Security and Information Intelligence Research
Oak Ridge National Laboratory, Oak Ridge, TN
lingerr, pleszkochmg, prowellsj, sayrekd@ornl.gov

Abstract

Software systems exhibit massive numbers of execution paths, and even comprehensive testing can exercise only a small fraction of these. It is no surprise that systems experience errors and vulnerabilities in use when many executions are untested. Computations over the functional semantics of programs may offer a potential solution. Structured programs are expressed in a finite hierarchy of control structures, each of which corresponds to a mathematical function or relation. A correctness theorem defines transformation of these structures from procedural logic into non-procedural, as-built specifications of behavior. These computed specifications enumerate behavior for all circumstances of use and cover the behavior space. Automation of these computations affords a new means for validating software functionality and security properties. This paper describes theory and implementation for loop behavior computation in particular, and illustrates use of an automated behavior computation system to validate a miniature looping program with and without embedded malware.

1. Computing the behavior of software

Much effort has been devoted to technologies and processes for software testing, resulting in substantial improvements in capabilities. However, testing faces a fundamental limitation: systems exhibit a massive number of possible execution paths, of which even the best testing can exercise only a small fraction. Errors and vulnerabilities can persist in field use because most executions are of necessity untested.

It is worth asking if knowledge of the full behavior of software for verification and security analysis can be obtained through means that do not depend on execution coverage.

Recent developments in software behavior computation [6,8] based on denotational semantics [1,7,10] suggest that the answer may be yes. Computer programs are mathematical artifacts subject to mathematical analysis. In particular, the single-entry, single-exit control structures of sequential structured programs, namely, sequence, ifthenelse, whiledo, and their variants, correspond to mathematical functions or relations that define mappings of domains to ranges or inputs to outputs. Structured programs define an algebraic hierarchy of these structures. Leaf node control structures in the hierarchy produce local functional effects that can be propagated in procedure-free form to containing structures to determine their local functional effects, continuing in this manner until the overall function of a program has been derived.

A correctness theorem [7] defines function-equivalent transformations from fundamental procedural control structures into functional forms which represent their as-built specifications, as follows, for control structure $P$, operations on data $g$ and $h$, predicate $p$, and program function $f$:

\[ \text{sequence:} \]

The function of a sequence control structure $P: g; h$

is computed by function composition $f = [P] = [g; h] = [h] \circ [g]$

where square brackets denote the behavior of the enclosed operations and “$\circ$” represents the composition operator.

---

1 Notice: This manuscript has been authored by UT-Battelle, LLC, under Contract No. DE-AC05-00OR22725 with the U.S. Department of Energy. The United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this manuscript, or allow others to do so, for United States Government purposes. This submission was written by the author(s) acting in their own independent capacity and not on behalf of UT-Battelle, LLC, or its affiliates or successors.
if
dense: The function of an if
dense next
P: if p then g else h endif
is computed by case analysis of true and false
branches
f = [P] = [(if p then g else h endif]
= ([p] = true → [g]) ∪ ([p] = false → [h])
where “∪” means the union of disjoint cases.

while
ndo: The function of a terminating while
ndo
P: while p do g enddo
is computed through function composition and
case analysis in a recursive equation based on
the equivalence of an iteration control struc-
ture and an iteration-free control structure (an
if then else):

\[ f = [P] = [\text{while } p \text{ do } g \text{ enddo}] = ([p] = true \rightarrow [g]) \cup ([p] = false \rightarrow [h]) \]

Computation of loop behavior is subject to theoretical
constraints as expressed in the Halting Problem. In
addition, while it always produces a correct result, this
recursive functional form is not readily understandable,
and must be augmented with an alternate approach. A
detailed description of loop behavior computation is
provided in the following section.

The theoretical foundations defined by the correct-
ness theorem have been applied in an emerging tech-
nology known as Function Extraction (FX), and have
been automated in a behavior computation system that
has been successfully applied to malware analysis
[11,12]. The FX system computes behavior for Intel
assembly language programs based on a repository of
functional semantics for X86 instructions. Behavior
computation at the assembly level, downstream from
compiler ambiguities, permits approaching the ground
truth of operations on the processor. In addition, the
behavior computation system employs a repository of
specification units known as semantic reduction theo-
rems (SRTs) that express computed behavior in func-
tion-equivalent expressions at higher levels of abstrac-
tion. SRTs are very general and can be defined once
and for all. For example, a repository of SRTs for fi-
nite arithmetic operations need never be changed unless
the processor architecture is modified.

The process of behavior computation proceeds
through four principal steps:

1. Transform input program instructions into
functional semantics form.

2. Transform the input program into structured
form expressed in an algebraic hierarchy of
fundamental control structures.

3. Compute and propagate the behavior of each
control structure beginning with leaf structures.

4. Apply semantic reduction theorems to simplify
and abstract computed behavior at each step.

Computed behavior is expressed in sets of disjoint,
non-procedural conditional concurrent assignments
(CCAs) that define conditions under which assignments
from domain to range occur. For example, behavior of
the simple sequence

\[ a := b + c \\
 b := b - a \\
 a := b - a \]

computed by simple composition, can be expressed as

\[ true \rightarrow a := -b - 2c \]
\[ b := -c \]

where the true predicate indicates the behavior always
occurs, and the assignments are concurrent mappings
from initial state into final state.

2. Loop behavior computation

As noted, loop behavior computation is subject to
theoretical constraints. However, research has resulted
in means to make the effects of these constraints arbi-
trarily small. In illustration, consider the loop com-
putation process depicted in five steps in Figure 1.

At a point where behavior is to be computed for a
loop, note that the loop body will be represented by a
non-procedural function previously computed based on
the structures and operations it contains. At step 1, the
loop is decomposed into a control slice loop that in-
cludes all operations associated with the iteration and
termination of the loop, and auxiliary slice loops, one
for each state item that is assigned new values in the
loop. At this point, the functional effect of the original
loop is equivalent to the union of the functional effects
of the sliced loops.

At step 2, the control slice loop is analyzed by se-
matic reduction theorems (SRTs) to determine how
the loop iterates and terminates. A class of SRTs em-
ployed in loop computation embodies iteration strate-
gies such as count up and terminate, count down and
terminate, terminate on condition, etc. As noted above,
SRTs are very general and broadly applicable units of
specification. For example, an SRT that embodies a
count down strategy will apply to count down behavior
that was computed from procedural logic, no matter
what particular implementation of counting down was
employed. This is an important characteristic based on
the property of “one function, many rules.” For example,
there are many ways to swap two variables in pro-
cedural logic, but, side effects aside, only one function
required to define them all, namely, “swap.” The net
effect of Step 2 is a derivation of the course of values
employed by the control slice to manage successive
iterations and eventual termination of the loop. Step 3 applies the course of values determined in step 2 together with other SRTs to produce computed behavior for the auxiliary slices whose iterative properties are now known. Step 4 combines the functional behaviors of the sliced loops to arrive at the composite behavior of the original loop. Finally, step 5 employs additional SRTs that may be applicable to reduce the computed loop behavior to simpler form.

![Figure 1. The loop behavior computation process](image)

Figure 1 depicts a miniature example of loop computation based on the process described above. Step 1 slices the loop into control and auxiliary slice loops as shown. At step 2, the control slice function and associated course of values are determined by applying a semantic reduction theorem that embodies count down iteration. Step 3 applies the course of values to the auxiliary slice loop to determine its function. Step 4 combines the control and auxiliary slice functions into a single conditional concurrent assignment specification. Finally, step 5 applies other SRTs to reduce the behavior to simpler terms. Note the box in the lower right of the figure that illustrates the effect of an initialization assignment on the computed behavior of loop. The initialization creates a sequence structure that can be composed to arrive at a simpler expression of behavior. Initialized loop behavior is often simpler than loop behavior alone.

### 3. A loop behavior computation example

Figure 3 shows a small subroutine in x86 assembly language that loops through elements in a memory array and counts the number of strictly positive elements. The byte before the start of the array is pointed to by the ESI register, and the ECX register is used to index into a particular array element. The computation starts at the end of the array and works backwards, so the ECX register counts downward. The EDX register is initialized to zero outside the loop, and is used to accumulate the number of strictly positive array elements. Internal to the loop body, the EAX register is loaded with the array element, and then the EBX register is set to 1 if the EAX value is strictly positive and set to 0 otherwise, and then the EBX register is added into the EDX register.

```
start:  xor EDX, EDX
top:    dec ECX
        jz done
        xor EBX, EBX
        mov EAX, [ESI+ECX]
        cmp EAX, 0
        setg BL
        add EDX, EBX
        jmp top
done:   ret
```

Figure 3. A miniature loop program
Figure 2. A loop behavior computation example

The structured version of this program produced by the FX system is shown below, expressed in terms of sequence, ifthenelse, and whiledo control structures:

```plaintext
xor DWORD EDX, DWORD EDX
label = 0x00401002
WHILE
  label equal 0x00401002
DO
  dec DWORD ECX
  IF
    jz BYTE 15
    THEN
      ret
      label = exit
  ELSE
    xor DWORD EBX, DWORD EBX
    mov DWORD EAX, DWORD [0+(ESI+(1*ECX))]
    cmp DWORD EAX, BYTE 0
    setg BYTE BL
    add DWORD EDX, DWORD EBX
    jmp BYTE -18
    label = 0x00401002
  END IF
END WHILE
```

The control slice for this loop program is as follows. It simply counts down the value in the ECX register:

```plaintext
WHILE
  label equal 0x00401002
DO
  dec DWORD ECX
  IF
    jz BYTE 15
    THEN
      ret
      label = exit
  ELSE
    xor DWORD EBX, DWORD EBX
    mov DWORD EAX, DWORD [0+(ESI+(1*ECX))]
    cmp DWORD EAX, BYTE 0
    setg BYTE BL
    add DWORD EDX, DWORD EBX
    jmp BYTE -18
    label = 0x00401002
  END IF
END WHILE
```

There are three pre-defined SRTs associated with recognizing the functional behavior of this control slice. First, a domain SRT states this loop always terminates regardless of the initial value of ECX. This is true because of the wrap-around nature of finite arithmetic. If the initial value of ECX is zero, then the loop will iterate 2^32 times. Second, there is a function SRT that states that the final value of ECX after the loop is 0. Finally, there is a course-of-values SRT that states that the values ECX takes on during the course of loop
iterations is downto_wrap_32(ECX,1), where downto\_wrap_32 is the 32-bit finite arithmetic wrapping version of the downto() list micro-operation illustrated in Figure 2.

The auxiliary slices are different than might be expected from looking at the code. Since the loop body code computes EAX from ECX, then EBX from EAX, and then EDX from EBX (and the prior EDX), it might seem as though the auxiliary slices should work in the same way. However, in loop behavior computation, the slicing is based on the previously computed functional behavior of the loop body, not the code of the loop body. Thus, since the value of EBX is computed from the newly assigned value of EAX which was computed from ECX, from the point of view of functional behavior, the auxiliary slice for EBX depends only on the control slice featuring only ECX. Similarly, the auxiliary slices for all the other loop variables (EDX, EAX, flags) only depend on the control slice.

Two auxiliary slice function SRTs are used to compute the remainder of the loop behavior from the control slice course-of-values. The first such SRT applies to all the loop variables except EDX, and states that if an auxiliary slice re-computes the auxiliary loop variable directly from the control slice every time, then its final value only depends on the last iteration of the loop, or on its original value if the loop never iterates. Thus, the final value of EAX is given by the following behavior expression computed by the FX system, expressed as a conditional concurrent assignment, where $M$ represents memory (\(|\) means the union of disjoint cases):

\[
\begin{align*}
\{ \text{ECX} == 1 \} & \rightarrow \\
EAX & := EAX \\
\{ \text{ECX} != 1 \} & \rightarrow \\
EAX & := M[\text{ESI} +d \text{d 1}] \\
\end{align*}
\]

Similarly, the final value of EBX is given by the following expression computed by the FX system:

\[
\begin{align*}
\{ \text{ECX} == 1 \} & \rightarrow \\
EBX & := EBX \\
\{ \text{ECX} != 1 \} & \&\& (M[\text{ESI} +d \text{d 1}] >s 0) \rightarrow \\
EBX & := 1 \\
\{ \text{ECX} != 1 \} & \&\& (M[\text{ESI} +d \text{d 1}] <=s 0) \rightarrow \\
EBX & := 0 \\
\end{align*}
\]

In the EBX behavior, the first case (ECX == 1) occurs when the loop body never executes, and EBX is left unchanged. The second case occurs when the loop body executes, and the last array element that it checks is strictly positive. The third case occurs when the loop body executes, and the last array element that it checks is zero or negative. The “+d” notation reflects 32-bit wrap-around addition, and the “>s” and “<=s” notations reflect a signed comparison instead of an unsigned comparison.

The second auxiliary slice function SRT applies only to the EDX auxiliary slice, which is recognized as accumulating the sum of an expression across the loop iterations, examining each array element in turn. For EDX, the final value (for the initialized loop) is given by:

\[
\{ EDX := \\
\text{sum of} \\
(x -> \text{is_pos_signed_32}(x)) \\
\text{over} \\
M[\text{ESI} +d \text{d 1}..(\text{ESI} +d \text{ECX} -d \text{d 1})] \\
\}
\]

Next, Figure 4 depicts an altered version of the loop, where a malicious code exploit has been deliberately added. Here, a check has been inserted into the loop body that looks for a -83 value in the memory array, and if that element value is found, the direction flag DF is set so as to change the direction of any future Intel string operations, which will likely result in a buffer overflow.

```
start:
xor EDX, EDX
top:
dec ECX
jz done
xor EBX, EBX
mov EAX, [ESI+ECX]
cmp EAX, 0
setg BL
add EDX, EBX
add EAX, 83
jnz top
std
ejm top
done:
ret
```

**Figure 4. Loop program with malicious code**

The FX system computes the behavior of this loop using slicing as usual. The control slice is exactly the same as before, and the behavior for ECX is likewise identical. The computed behavior for EAX is different, reflecting the fact it is used as for temporary storage in the malicious exploit:

\[
\begin{align*}
\{ \text{ECX} == 1 \} & \rightarrow \\
EAX & := EAX \\
\{ \text{ECX} != 1 \} & \rightarrow \\
EAX & := M[\text{ESI} +d \text{d 1}] +d 83 \\
\end{align*}
\]
The new behavior for EBX and EDX is the same as in the original code, because the auxiliary slices are the same. Since the EDX register holds the final answer of the computation, testing that only checks the value of EDX will not find the malicious exploit.

However, this time, the FX computation shows additional behavior for a new auxiliary slice for the DF flag which reflects the fact that it is set under certain conditions. If there is an auxiliary slice function SRT to specifically recognize these conditions, then the added behavior will be shown as follows:

```
[ exists
  (x -> (x == -83))
over
  M[(ESI +d 1)..<ESI +d ECX -d 1)]
  -> DF := true
| not (exists
  (x -> (x == -83))
over
  M[(ESI +d 1)..<ESI +d ECX -d 1)]
  -> DF := DF
]
```

Otherwise, if there is no SRT to specifically recognize these conditions, as would typically be the case, then the behavior of DF in the conditional concurrent assignment statement will be expressed in terms of recursively defined generic loop behavior functions as computed through direct application of the correctness theorem. However, even in this situation, it is clear that undesired code has been inserted, as there is no reason to set the DF flag under any circumstances.

This is a key point. As depicted in Figure 5, testing can cover only a fraction of the execution space, but behavior computation covers all of the behavior space. That is, the disjoint conditional concurrent assignments produced by behavior computation represent all possible program behaviors, whether correct or incorrect, legitimate or malicious.

With respect to this example, it is unlikely that a test case would be created to execute the program under the condition that a -83 value was present in the array. For an intruder who inserted the malicious code, however, this is the attack vector for achieving desired objectives. But for the FX system, this is just another case of computed behavior that will be automatically produced along with the others.

This example illustrates the potential power of behavior computation to help verify correct functionality and security properties, beyond what is practically achievable with testing. Of course, other forms of testing will always be required, for example, integration and performance testing, but computed behavior may help to improve the speed and completeness of verification, particularly at unit and subsystem levels.

![Figure 5. Testing and behavior computation](image)

4. Computed exploration of software

The behavior computation process for a program requires deriving the behavior of every control structure of which it is comprised. This rich body of highly structured behavioral sub-specifications associated with control structures can be applied to exploration of code in behavioral terms [2,3]. Such exploration provides fast insight into how particular behaviors are produced. In particular, three methods for applying computed behavior to program exploration have been investigated in the FXplorer application, as follows:

- BehaviorCase or Path Quest
- BehaviorHere or Come Here
- BehaviorPath or Connect the Dots

By default, FX displays the whole program behavior database. Using this display, a user might decide that one or more of the behaviors looks suspicious or erroneous. The user might want to know which code statements and their accumulating behaviors contribute to the case in question.

BehaviorCase, FXplorer’s Path Quest function, starts with a user-selected case in the behavior database of a program. It determines and displays the compositions of all the accumulating behavior along all the code paths that produce that case. All other code and behavior is eliminated. Thus, a programmer can determine what part of the original program is responsible for a given result. Figure 6 shows an example of the Path Quest function and accumulating behavior.
BehaviorHere, or FXplorer’s *Come Here* function, starts with a user-selected statement in the program. It determines and displays the compositions of all the accumulating behaviors along all possible code paths to that statement. *Come Here* allows a programmer to identify a particular point in a program and see all the paths and accumulating behaviors leading to that point. Figure 7 shows a *Come Here* function performed on statement 6 of Figure 6.

![Figure 6. BehaviorCase or *Path Quest* exploration](image_url)

BehaviorPath, or *Connect the Dots*, starts with a user-selected code path through the program. It determines and displays all the compositions of the accumulating behavior along that path. By connecting the dots, a programmer can examine a particular path through the program to see the accumulating and final behavior it causes. Figure 8 shows user controlled connect-the-dots exploration.

![Figure 7. BehaviorHere or *Come Here* exploration](image_url)
These three functions provide a unique way of understanding a program. They allow direct answers to common programmer questions such as: "Where does this result come from?" (BehaviorCase), "What happens if this path is executed?" (BehaviorPath), and "How does this program get here?" (BehaviorHere). The ability to answer these questions in full without doing a line-by-line analysis improves a programmer’s ability to understand program behavior, to verify that the results are correct, and to validate the results against documentation of specifications or designs.

Capabilities such as these can provide programmers and analysts with access to computed behavior in the context of immediate needs for program understanding, verification, debugging, modification, or evolution.

Figure 8. BehaviorPath or Connect-the-Dots exploration

5. Comparison with other work

The technology of behavior computation can be contrasted with symbolic execution and abstract interpretation as follows:

5.1 Symbolic execution

Symbolic execution [5] is an approach originating in the 1970’s to understand what a program does on generalized input data expressed using symbolic variables. Typically, symbolic execution only traces through a single possible execution path of a program. Recently, symbolic execution has been generalized to operate over a limited subset of related execution paths, e.g., paths that differ only in the number of times a given loop is iterated [9]. Behavior computation significantly generalizes symbolic execution by expressing the complete behavior of a program over all possible paths as a static, symbolic expression. This is possible because behavior computation uses automated program structuring to recover the underlying hierarchical structure of a program, so that all its control structures, which embody all its paths, can be analyzed in a systematic manner. In behavior computation, a program is examined in a bottom-up process, first analyzing the leaf-node control structures, then combining those analyses at higher and higher levels to understand the larger program. In symbolic execution, paths are examined starting from the entry point of the program, and it is a difficult, heuristic effort to combine the results from different paths. Programs can exhibit massive numbers of possible execution paths, but are always comprised of a finite number of control structures. Behavior computation is a finite process that deals with control structures, not paths, and is thus guaranteed to terminate.

5.2 Abstract interpretation

Abstract interpretation [4] is an application of denotational semantics in which a scaled-down, computationally tractable semantic domain is used to rigorously approximate a fuller, mathematically defined but computationally intractable semantic domain. In cases where the scaled-down domain of abstract interpretation is still suitable for an application, the extraction
6. Next steps in behavior computation

Oak Ridge National Laboratory (ORNL) is applying behavior computation to smart grid components, with initial focus on embedded software in smart meters. The computations are based on the functional semantics of instructions for the MSP430 processor, plus functional definitions for meter components under software control. The objective of this effort is to analyze computed behavior both to determine if functionality is correct and to help reveal any security vulnerabilities that may be present.

Research and development in software behavior computation will continue at ORNL to exploit this new technology in other areas, such as software development, test and evaluation, reverse engineering, and anti-tamper analysis. FX algorithms are being migrated to high performance computing (HPC) environments available at ORNL.

Application of the technology to analysis of software security properties is a focus area [13]. Malware exhibits a fundamental vulnerability: it must execute on the target computer in order to achieve its objectives, and the executable code unavoidably embodies computable behavior, just as does any other code. FX computes the behavior of all of it, whether legitimate or malicious. In so doing, it reveals the presence of any malicious content, for example, sleeper code or logic bombs, that may be hidden in legitimate code.

7. References


